Certification of SAT Solvers in Coq

Jean-Pierre Jouannaud, Pierre-Yves Strub, and Lianyi Zhang

INRIA-Tsinghua Project FORMES

INRIA-LIAMA and Tsinghua University, Beijing

Abstract. We describe here a fully portable, open source certifier for traces of SAT problems produced by zChaff [6][1]. It can also be easily adapted for MiniSat [4][2], PicoSat [2][3] and Booleforce[4], which we have done for PicoSat. Our certifier has been developped with the proof assistant Coq[5]. We give some figures based on the pigeon hole, comparing both PicoSat and zChaff on the one hand, and our certifier with another certifier developped with Coq.

1 Introduction

The importance of SAT solvers has grown over the years in verification, for two major reasons: because *bit blasting* allows to transform many verification problems in a formula of propositionnal logic ; and because SAT solvers have made enormous progress, making them efficient for many practical *problems* which are indeed seldom in the NP-hard area.

But the sophistication of modern SAT solvers, usually written in C for efficiency, usually not proved correct because of the size and the sophistication of the code, makes them error-prone. Stories are many, and well-known. On the other hand, most modern SAT solvers do not output a bare Yes/No answer, but a trace which is an easily checkable satisfying assignement in the positive case, and can be interpreted as a resolution proof in the negative case. The idea has therefore been around for a while to certify such a trace with a theorem prover in order to be confident that the output, in the case of a negative answer, is indeed a valid trace. This seems a rather trivial problem, but it is indeed not. On the one hand, proving a program correct is *never* a trivial task even with the modern provers on the market. On the other hand, and this is the real difficulty, bit blasting produces large inputs, which in turn produce very large traces, measured in terms of hundred of megabites, or even gigabites. A consequence is that such simple things

[1] zChaff can be doawnloaded at url http://www.princeton.edu/~chaff/zchaff/index1.html

[2] MiniSat can be doawnloaded at the url http://minisat.se/MiniSat+.html

[3] PicoSat can be downloaded at url http://fmv.jku.at/picosat/

[4] The Booleforce website is at url http://fmv.jku.at/booleforce

[5] The Coq website is at url http://coq.inria.fr/

as parsing the trace within the theorem prover becomes an impossible task. And of course, the proof object obtained from the trace becomes even larger, making proof-checkig extremely delicate. Certifying the trace obtained by a SAT solver is a quite non-trivial task, which requires expertise in using the chosen theorem prover.

We are aware of several attempts of solving this problem by using the `Coq` proof assistant, and there has been other attempts with various provers, in particular with HOL [6], and Isabelle [7]. A first approach is to avoid reflection and code SAT directly in `Coq` [5]. The second was not very conclusive, and the `Coq` development is not available [3]. The third was very conclusive, and the obtained development is available [1]. However, in order to process very large traces, the authors had to compromise the `Coq` kernel by introducing impure non-fonctional features in `Coq` in order to make proof-checking more efficient *for their application*. It follows that the proof-term produced by their code cannot be proof-checked by the official `Coq` kernel.

In this paper, we describe our own certifier, which produces a trace checkable by `Coq` 8.2, without using any impure extension of `Coq` (we do use machine integers, available in `Coq` 8.2). We believe that our certifier is the first efficient, *portable* certifier for both `PicoSat` and `zChaff` which were our targets. Its efficiency is comparable to that of [1], although a little bit less. It would be possible to optimiwe it, still, and approach the efficiency of [1], to the price of a less elegant development. So far, we resisted following this path.

In the following, we describe first the problem format in Section 2, the trace formats of `zChaff` and `PicoSat` (`MiniSat` has a very similar one) in Section 3, the encoding in `Coq` in Section 4, and the checking itself in Section 5. We conclude in Section 7.

2 Problem format

The problem file format is the standard DIMAC format.

A DIMAC file is line oriented. It starts with comments, i.e. with lines beginning with the letter *c*. Then the number of variables and clauses may be defined by a line beginning with the letter *p* followed by the word *cnf*, the number of variables, and the number of clauses. Then, each of the next lines describes a clause: they are composed of a sequence of non null integers, followed by the integer 0. Each non-null integer defines a literal. For `PicoSat`, a positive (resp. negative) integer n defines the positive literal x_n (resp. the negative literal $\neg x_n$).

For a simple example, the problem $\{x_1 \vee x_2, \neg x_1, x_1 \vee \neg x_2\}$ can be defined by the following DIMAC file:

```
c My Simple PicoSat Problem
p cnf 2 3
1 2 0
```

[6] The HOL website is at url
`http://www.cl.cam.ac.uk/~jrh13/hol-light`

[7] The Isabelle website is at url
`http://www.cl.cam.ac.uk/research/hvg/Isabelle`

```
-1 0
1 -2 0
```

zChaff uses a slightly different encoding: the positive literal x_n is encoded as $2n$, whereas the negative literal $\neg x_n$ is encoded as $2n + 1$. The same example becomes:

```
c My Simple zChaff Problem
p cnf 2 3
1 4 0
3 0
1 5 0
```

In real practical problems, there may be thousands of litterals and clauses.

3 Trace formats

3.1 zChaff

zChaff uses a line oriented trace. The trace is split into 3 parts: a (possibly empty) set of learned clauses, a (possibly empty) set of variable assignments and a conflict description. Clauses of the problem are not part of the trace.

Learned clauses The first part of the trace defines a set of learned clauses. For example, a line of the form

```
CL: 9 <= 6 1 0
```

indicates that a new clause, indexed by 9, can be deduced by resolutions of the clauses indexed by 6, 1 and 0, that is, by first resolving the clauses indexed by 6 and 1, and then resolving the resolvent with the clauses indexed by 0. The clauses 6, 1 and 0 are called the *antecedent* of 0.
All clauses of the problem are indexed by starting from the index 0 for the first clause. Clauses that do not appear in the trace are indexed as well.

Note that the final resolvent is not given, nor the pivot variables used in the resolutions. zChaff ensures that the result does not depend on the chosen pivot variable.[8] Only the resolution order - from left to right - is given.
In the zChaff implementation we used, the learned clauses appear in the correct order, i.e. the antecedents of a learned clauses are either clauses of the problem or learned clauses appearing above in the trace. This is a major practical difference with PicoSat.

Variables assignments Then come variables assignments. Each variable assignment is described by a line of the form:

```
VAR: 3 L: 5 V: 1 A: 1 Lits: 6 8
```

[8] Similarly to Booleforce, zChaff must use linear resolution, but we were not able to confirm this point.

This tells us that the variable x_3 (`VAR: 3`) must be assigned to true (`V: 1`). This is because of the clause of index 1 (`A: 1`), which literals are x_3 and x_4 (`Lits: 6 8`). We also know that this assignment has been made at the decision level 5 (`L: 5`).

If the clause 1 is - as stated by the trace - $x_3 \vee x_4$, then, for this clause to be true, one of the variables x_3 and x_4 must be assigned to true. Hence, there must exist a variable assignment with a lower decision level assigning false to x_4. At some point, the trace should contain variables assignments which antecedents are unit clauses, and which decision levels are minimal, that is, a variable assignment of the form

```
VAR: 5 L: 0 V: 0 A: 9 Lits: 11
```

Here, the clause of index 9 is of the form $\neg x_5$, implying that the variable 5 must be assigned to false.

Conflict The conflict is a single line of the form:

CONF: 3 == 3 7

which indicates that a conflict comes from the clauses 3, which literals are $\neg x_2$ and $\neg x_4$. Therefore, there must be variables assignments assigning true to x_2 and x_4, hence implying the non-satisfiability of the clause indexed by 3.

3.2 `PicoSat`

The traces are quite similar in spirit, with the exception that clauses are not sorted in the output trace of `PicoSat`: a clause index may be used before the clause is indeed defined. Since the ordering of clauses is crucial in the `Coq` proof, the output trace must be ordered before the proof-term can be generated. The `Coq` libraries provide with various, efficient, proved sorting algorithms. However, they slow down the whole proof checking process which makes comparisons between `zChaff` and `PicoSat` almost senseless. We have therefore hardcoded the sorting algorithm in the reader of the `PicoSat` output trace.

4 Encoding of the problem into `Coq`

Inefficient, easy to reason about encoding We first defined all the necessary data structures for reasoning about SAT problems in `Coq`. Here, the aim is not to have data structures for computing efficiently, but to have data structures that fit with our need to reason about. For example, a literal is simply defined as a pair of a positive integer (the variable index) and a boolean (the sign of the literal). Likewise, a clause is a list of literals, and a SAT problem a list of clauses. The valuation is then simply defined as a function from integers to boolean, and we defined the interpretation of clauses (or set of clauses) by list folding. The notion of satisfiability and validity is then defined with respect to this notion of interpretation.

With these structures, we can prove that the resolution rule is correct with respect to the interpretation function.

Data structures for computation We then defined the same data structures, but in a more efficient way. For example, Peano integers are replaced by inductive binary integers or machine binary integers. We defined a translation function from this representation to the previous one. The interpretation function of this clauses representation is defined as the interpretation of their translation to the previous representation.

Reflection We also defined a reflection function translating the inefficient representation of clauses to the propositional level of `Coq`. We proved that they are equivalent, that is, that an encoded clause is valid if and only if its translation to the proposition level of `Coq` is a provable Lemma.

Encoding of the trace For this encoding, we use the previously defined efficient encoding of clauses. The set of initial clauses is stored into a Patricia tree, where the indexes of the clauses are the key used in the tree. The trace is a list of clauses to be learned. Each element of this list is a pair composed of the index of the clause and of the list of its antecedents. The variables assignments, and the conflict description, are encoded as learned clauses, an encoding to be explained later.

Trace verification We defined a function taking a set of initial clauses (the Patricia tree previously described) and a trace, and returning the set of all the learned clauses (along with the initial ones). This final set is simply obtained by iterating a function on the trace which take the antecedents, find the corresponding clauses, compute the final resolvent and adds this resolvent to the Patricia tree. In case of an error (one of the antecedent clauses cannot be found, there are no possible resolutions, ...), an error value is returned.

How to compute the resolvent If the two clauses to be resolved are sorted, one of the resolvent can be efficiently computed by a slight modification of the merge function (of the merge-sort algorithm) which must discard the first pair of opposite literals it encounters. Note that the resolvent is then also sorted. As a consquence, we need to sort the set of initial clauses.

Adequation Finally, if for a given problem and trace, the final set of learned clauses contains the empty clause, then we can prove that the translation of the problem into the propositional level of `Coq` implies False, implying that the initial problem is not satisfiable.

5 Checking a trace in practice

We start from a SAT problem and a `zChaff` trace.

- We first generate a `Coq` Lemma representing the UNSAT problem. E.g., from the DIMAC file given above, we generate the following `Coq` Lemma:

```
Definition UNSAT : Prop := forall (X : nat -> Prop),
  (X 1) \/ (X 2) -> ~(X 1) -> (X 1) \/ ~(X 2)
  -> False.
```

- We then post-process the `zChaff` trace, removing the variables assignments and the conflict descriptions. Note that a variable assignment is simply the encoding of a unit clause. For example, the variable assignment

```
VAR: 3 L: 5 V: 1 A: 1 Lits: 3 6 8 10
```

is translated into the unit clause x_3, which can be learned by resolving the antecedent 1 with the unit clauses related to the literals $\neg x_1$, x_4 and x_5, that is the literals 3, 8 and 10 in the encoding. Note that the literal 6 is not considered as it corresponds to the variable 3, which is the target variable of this variables assignment. Likewise, the conflict is encoded as the empty clause.
- We load the problem and the trace using a dedicated `Coq` tactic.
- We can then compute the set of learned clauses and check that the empty clause is part of this set.
- Using the adequation Lemma, we conclude that the UNSAT Lemma defined at the very beginning is a valid `Coq` Lemma.

6 Comparisons

The figures will be presented during the talk.

7 Conclusion

Acknowledgements: Our colleague professor Bow-Yaw Wang helped us understand learning in `PicoSat` by rewriting it in `Caml`[9].

References

1. Michaël Armand, Benjamin Grégoire, Arnaud Spiwack, and Laurent Théry. Extending Coq with Imperative Features and its Application to SAT Verification. In *Interactive Theorem Proving*, Edinburgh Royaume-Uni, 2010.
2. Armin Biere. Picosat essentials. *JSAT*, 4(2-4):75–97, 2008.
3. Ashish Darbari, Bernd Fischer 0002, and João P. Marques Silva. Industrial-strength formally certified sat solving. *CoRR*, abs/0911.1678, 2009.
4. Niklas Eén and Niklas Sörensson. An extensible sat-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *SAT*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.
5. Stéphane Lescuyer and Sylvain Conchon. Improving coq propositional reasoning using a lazy cnf conversion scheme. In Silvio Ghilardi and Roberto Sebastiani, editors, *FroCos*, volume 5749 of *Lecture Notes in Computer Science*, pages 287–303. Springer, 2009.
6. Yogesh S. Mahajan, Zhaohui Fu, and Sharad Malik. Zchaff2004: An efficient sat solver. In Holger H. Hoos and David G. Mitchell, editors, *SAT (Selected Papers*, volume 3542 of *Lecture Notes in Computer Science*, pages 360–375. Springer, 2004.
7. Tjark Weber and Hasan Amjad. Efficiently checking propositional refutations in hol theorem provers. *Journal of Applied Logic*, 7(1):26 – 40, 2009. Special Issue: Empirically Successful Computerized Reasoning.

[9] The `Caml` website is at url `http://caml.inria.fr/download.en.html`